

FIG. 1

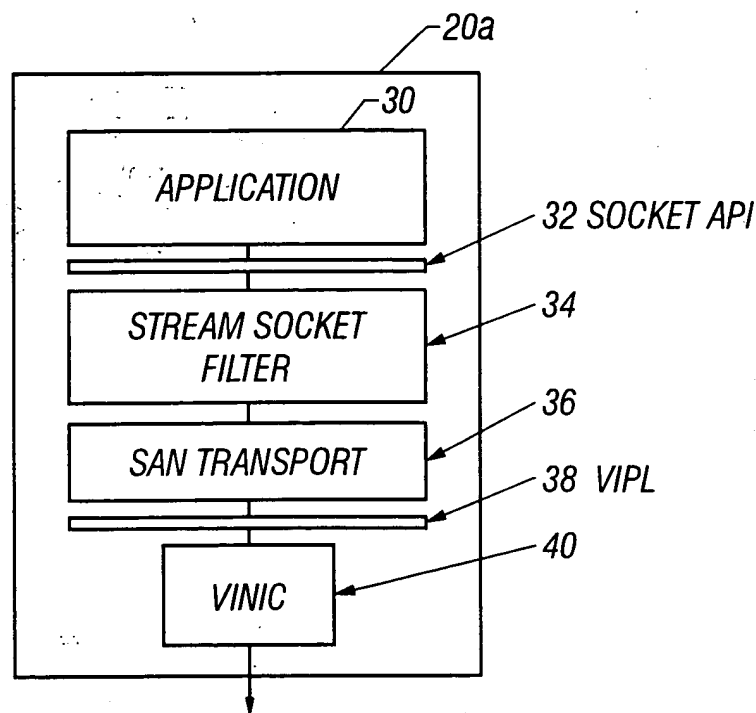


FIG. 2

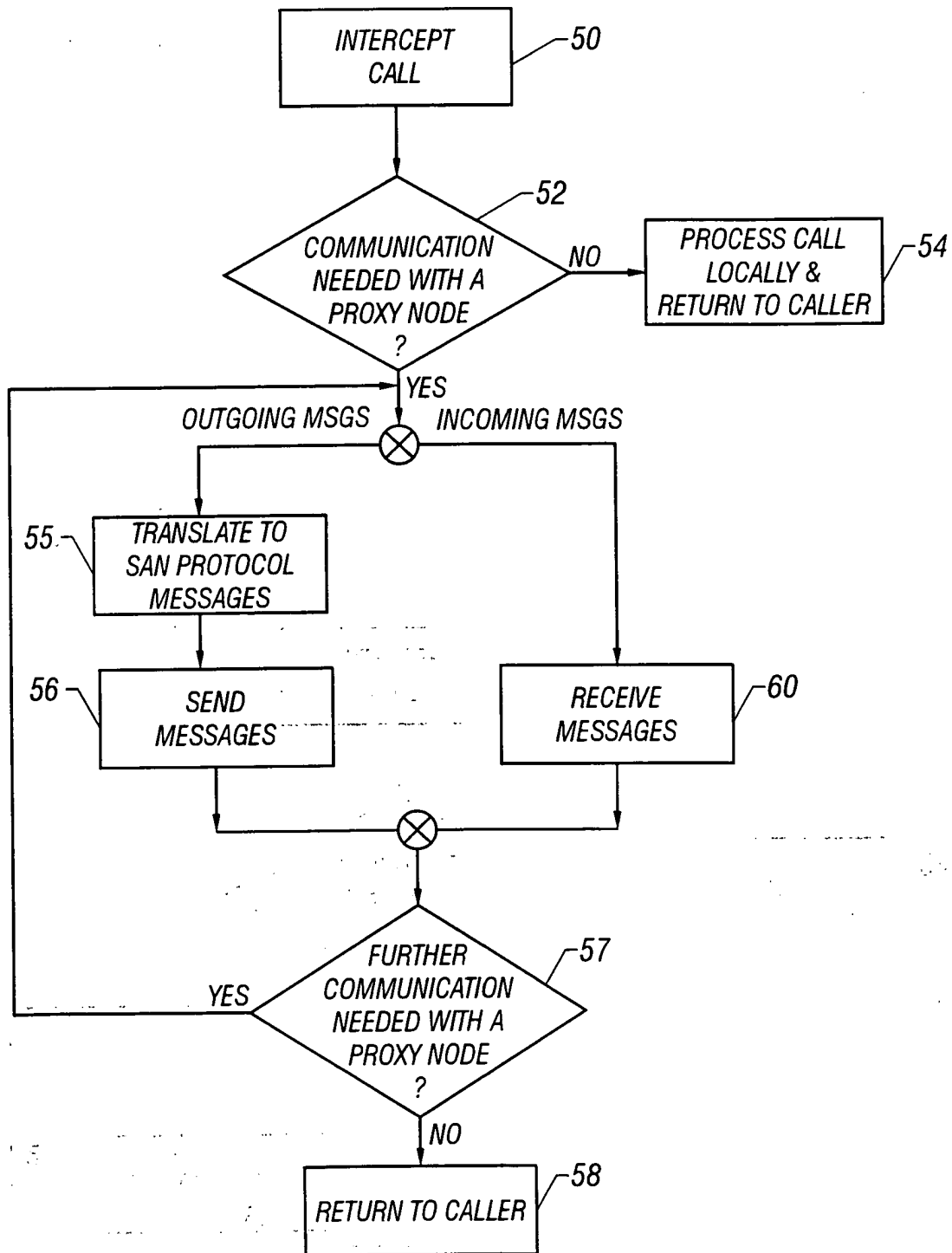


FIG. 3

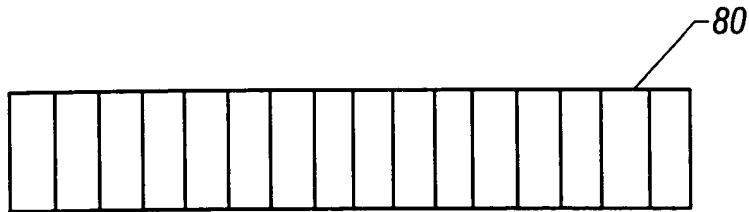


FIG. 4A

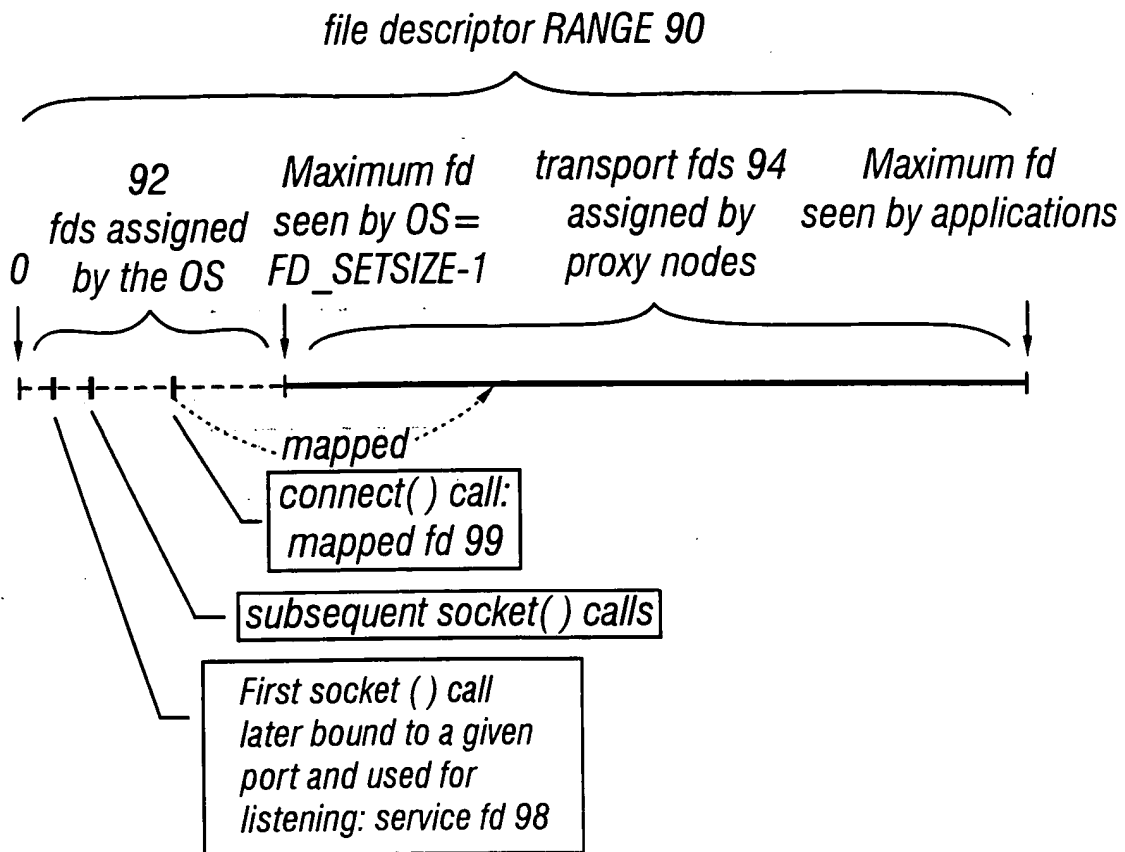


FIG. 4B

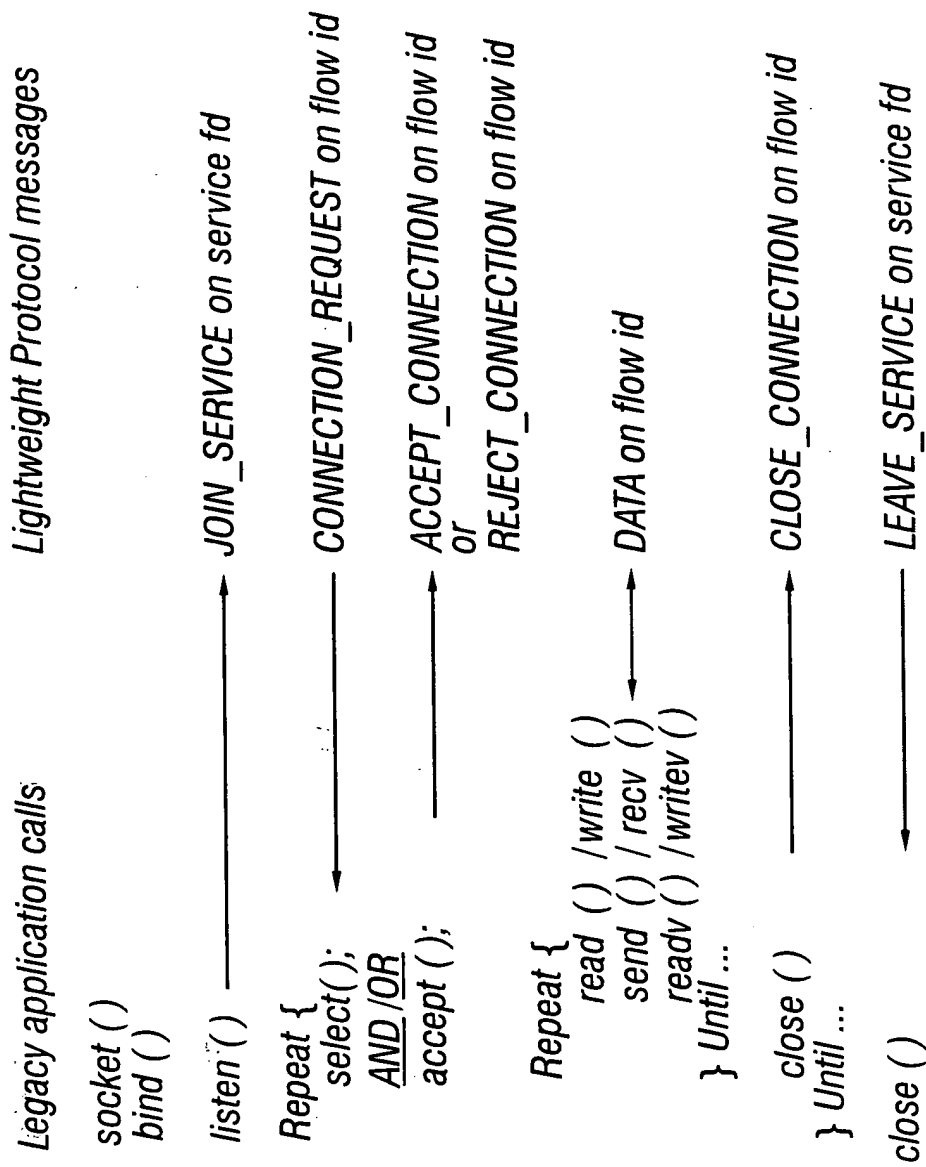


FIG. 5A

Message Type	Description
JOIN_SERVICE	Sent by an application node when joining a group of service offered by SAN proxies.
LEAVE_SERVICE	Sent by an application node when leaving a group of service offered by SAN proxies.
SHUTDOWN_SERVICE	Sent by a SAN proxy when it shuts down a service
CONNECTION_REQUEST	Sent by a SAN proxy with flow identifier to an application node indicating that the proxy received a connection request from a client. Also, sent by an application node to actively open a connection.
ACCEPT_CONNECTION	Sent by an application node (SAN proxy) to positively acknowledge to a SAN proxy (application node) regarding the acceptance of a connection request.
REJECT_CONNECTION	Sent by an application node (SAN proxy) to negatively acknowledge to a SAN proxy (application node) regarding a connection request.
CLOSE_CONNECTION	Sent by an application node (SAN proxy) to SAN proxy (application node) for closing a connection.
CREDIT_CONNECTION	Used to request credit information.
CREDIT_RESPONSE	Used to send credit information.
DATA	

FIG. 5B

```
socket() → sf_socket(domain, service, protocol) {  
    if (this is a TCP socket) {  
        if (called for the first time) {  
            perform SAN transport initialization;  
            Start up SAN Transport;  
            fd = socket (domain, service, protocol);  
            Note fd of first socket call;  
            return(fd);  
        }  
        else {  
            fd = socket (domain, service, protocol);  
            return(fd);  
        }  
    }  
    else  
        return (socket (domain, service, protocol) );  
}
```

FIG. 6A

```
bind() → sf_bind (fd, sockaddr, addrlen) {  
    Note IP address & port #;  
    if (this is a TCP socket) {  
        if (port is specified)  
            note fd as service fd for this port;  
        return (bind (fd, sockaddr, addrlen));  
    }  
    else  
        return (bind (fd, sockaddr, addrlen));  
}
```

FIG. 6B

```

connect() → sf_connect (fd, sockaddr, addrlen) {
    Note IP address & port #;
    if (this is a TCP socket) {
        if (this is a non-blocking socket) {
            if (CONNECTION_REQUEST msg not previously sent for this fd)
                send CONNECTION_REQUEST msg with fd to proxy node;
            if (ACCEPT_CONNECTION or REJECT_CONNECTION msg is pending) {
                if (receive ACCEPT_CONNECTION msg) {
                    assign mapped fd by mapping OS-assigned fd to a transport fd;
                    return (success);
                }
                else
                    return (connection refused error);
            }
            else
                return (connection in progress);
        }
        else {
            send CONNECTION_REQUEST msg with fd to proxy node;
            wait to receive (ACCEPT_CONNECTION or REJECT_CONNECTION msg);
            if (receive ACCEPT_CONNECTION msg) {
                assign mapped fd by mapping OS-assigned fd to a transport fd;
                return (success);
            }
            else
                return (connection refused error);
        }
    }
    else
        return (connect (fd, sockaddr, addrlen));
}

```

FIG. 6C

```
listen() --> sf_listen(fd, backlog) {  
    switch (type of fd) {  
        case service fd:  
            send JOIN_SERVICE msg;  
            return (success);  
        case mapped fd:  
        case transport fd:  
            return (exception error);  
        default:  
            return (listen(fd, backlog));  
    }  
}
```

FIG. 6D

```
accept() --> sf_accept (fd, clientaddr, len) {  
    switch (type of fd) {  
        case service fd:  
            if (this is a non-blocking socket) {  
                if CONNECTION_REQUEST msg is pending for this service fd {  
                    read CONNECTION_REQUEST msg with proxy-assigned flow id;  
                    if (connection can be accepted) {  
                        send ACCEPT_CONNECTION msg;  
                        return (flow id);  
                    }  
                    else {  
                        send REJECT_CONNECTION msg;  
                        return (try again);  
                    }  
                }  
            }  
            else  
                return (try again);  
        }  
    }  
    else {
```

FIG. 6E-1

```
while (1) {  
    if CONNECTION_REQUEST msg is pending for this service fd {  
        read CONNECTION_REQUEST msg with proxy-assigned flow id;  
  
        if (connection can be accepted) {  
            send ACCEPT_CONNECTION msg;  
            return (flow id);  
        }  
        else {  
            send REJECT_CONNECTION msg;  
        }  
    }  
    else  
        wait to receive CONNECTION_REQUEST msg;  
    } // while loop  
}  
  
case transport fd:  
    return (exception error);  
  
default:  
    return ( accept (fd, clientaddr, len));  
}  
}
```

FIG. 6E-2

```

select() → sf_select (nfd, readfds, writefds, exceptfds, timeout) {

  note the number of fds to select on;
  set timeslice as a function of timeout and number of fds;

  do forever {

    // PHASE 1: POLL ALL FDS
    for each service fd in readfds {
      if CONNECTION_REQUEST msg is pending for this service fd
        set corresponding service fd as available;
    }
    for each transport fd in readfds {
      if DATA msg is pending for this transport fd
        set corresponding transport fd as available;
    }
    for each mapped fd in readfds {
      perform mapping to transport fd;
      if DATA msg is pending for this transport fd
        set corresponding mapped fd as available;
    }

    for each transport fd in writefds {
      if DATA msg can be sent on this transport fd
        set corresponding transport fd as available;
    }
    for each mapped fd in writefds {
      perform mapping to transport fd;
      if DATA msg can be sent for this transport fd
        set corresponding mapped fd as available;
    }
  }
}

```

FIG. 6F-1

FIG. 6F-2

```

for each service fd in exceptfds {
  if exception occurs for this service fd
    set corresponding service fd;
}
for each transport fd in exceptfds {
  if exception occurs for this transport fd
    set corresponding transport fd;
}
for each mapped fd in exceptfds {
  perform mapping to transport fd;
  if exception occurs for this transport fd
    set corresponding mapped fd;
}

for all other fds
  call original system select();

combine all available descriptors;

if (one or more descriptors are ready)
  return (number of descriptors available);
else
  choose one descriptor in readfds to wait on; // heuristic-based choice
  restore original descriptor sets;

if (time is up AND no fd is available)
  return (timed out);

// PHASE 2: WAIT if necessary
wait for arrival of CONNECTION_REQUEST, ACCEPT_CONNECTION,
  REJECT_CONNECTION or DATA msg for the chosen descriptor, up to timeslice;
}

```

```

recv() → sf_recv (fd, buf, len, flags) {

  switch (type of fd) {

    case service fd:
      return (exception error);

    case mapped fd:
      perform mapping to transport fd;

    case transport fd:
      if (MSG_WAITALL flag is not set) {
        if at least one DATA msg is pending for this transport fd {
          receive data into buf;
          return (number of bytes read);
        }
        else {
          if (this is a non-blocking socket)
            return (resource not available);
          else {
            wait to receive a DATA msg for this transport fd;
            receive data into buf;
            return (number of bytes read);
          }
        }
      }
      else {
        wait until all len bytes of DATA msgs for this transport fd arrives;
        receive data into buf;
        return (number of bytes read);
      }
    }

  default:
    return ( recv (fd, buf, len));
  }
}

```

FIG. 6G

```
send() → sf_send (fd, buf, len, flags) {  
    switch (type of fd) {  
        case service fd:  
            return (exception error);  
        case mapped fd:  
            perform mapping to transport fd;  
        case transport fd:  
            if (this is a non-blocking socket){  
                if (no DATA msg can be sent at this time)  
                    return (try again);  
                else  
                    send DATA msg(s) with data from buf in non-blocking fashion;  
            }  
            else {  
                if ( no DATA msg can be sent at this time)  
                    Wait until atleast one DATA msg can be sent;  
                send DATA msg(s) with data from buf;  
            }  
            return (number of bytes sent);  
        default:  
            return (send (fd, buf, len));  
    }  
}
```

FIG. 6H

```
read() → sf_read (fd, buf, len) {  
    switch (type of fd) {  
        case service fd:  
            return (exception error);  
        case mapped fd:  
            perform mapping to transport fd;  
        case transport fd:  
            if at least one DATA msg is pending for this transport fd {  
                receive data into buf;  
                return (number of bytes read);  
            }  
            else {  
                if (this is a non-blocking socket)  
                    return (resource not available);  
                else {  
                    wait to receive a DATA msg for this transport fd;  
                    receive data into buf;  
                    return (number of bytes read);  
                }  
            }  
        default:  
            return ( read (fd, buf, len));  
    }  
}
```

FIG. 6I

```
write() → sf_write (fd, buf, len) {  
    switch (type of fd) {  
        case service fd:  
            return (exception error);  
        case mapped fd:  
            perform mapping to transport fd;  
        case transport fd:  
            if (this is a non-blocking socket){  
                if (no DATA msg can be sent at this time)  
                    return (try again);  
                else  
                    send DATA msg(s) with data from buf in non-blocking fashion;  
            }  
            else {  
                if( no DATA msg can be sent at this time)  
                    Wait until atleast one DATA msg can be sent;  
                send DATA msg(s) with data from buf;  
            }  
            return (number of bytes written);  
        default:  
            return (write (fd, buf, len));  
    }  
}
```

FIG. 6J

```
readv() → sf_readv (fd, vector_buf, vector_count) {  
    switch (type of fd) {  
        case service fd:  
            return (exception error);  
        case mapped fd:  
            perform mapping to transport fd;  
        case transport fd:  
            if at least one DATA msg is pending for this transport fd {  
                scatter data received into vector_buf;  
                return (number of bytes read);  
            }  
            else {  
                if (this is a non-blocking socket)  
                    return (resource not available);  
                else {  
                    wait to receive a DATA msg for this transport fd;  
                    scatter data received into vector_buf;  
                    return (number of bytes read);  
                }  
            }  
        default:  
            return ( readv (fd, buf, len));  
    }  
}
```

FIG. 6K

```
writev() → sf_writev (fd, vector_buf, vector_count) {  
    switch (type of fd) {  
        case service fd:  
            return (exception error);  
        case mapped fd:  
            perform mapping to transport fd;  
        case transport fd:  
            if (this is a non-blocking socket){  
                if (no DATA msg can be sent at this time)  
                    return (try again);  
                else  
                    send DATA msg(s) with gathered data from vector_buf;  
            }  
            else {  
                if( no DATA msg can be sent at this time)  
                    Wait until atleast one DATA msg can be sent;  
                send DATA msg(s) with gathered data from vector_buf;  
            }  
            return (number of bytes written);  
        default:  
            return (writev (fd, buf, len));  
    }  
}
```

FIG. 6L

```
ioctl() → sf_ioctl (fd, request, arg) {  
    switch (type of fd) {  
        case service fd:  
            return (socket not connected error);  
        case mapped fd:  
            perform mapping to transport fd;  
        case transport fd:  
            switch (request) {  
                case FIONBIO:  
                    set non-blocking I/O variable to value in arg;  
                    return (success);  
                case FIOASYNC:  
                    set async I/O variable to value in arg;  
                    return (success);  
                case FIONREAD:  
                    peek at DATA msg for this transport fd;  
                    set number of bytes in arg;  
                    return (success);  
                default:  
                    return (warning: option not meaningful in SAN Transport);  
            }  
        default:  
            return (ioctl (fd, request, arg));  
    }  
}
```

FIG. 6M

```
getsockname() → sf_getsockname (fd, localaddr, addrlen) {  
  
    switch (type of fd) {  
  
        case service fd:  
            return (socket not connected error);  
  
        case mapped fd:  
            perform mapping to transport fd;  
  
        case transport fd:  
            return (local protocol address associated with this transport fd);  
  
        default:  
            return (getsockname (fd, localaddr, addrlen));  
    }  
}
```

FIG. 6N

```
getpeername() → sf_getpeername (fd, localaddr, addrlen) {  
    switch (type of fd) {  
        case service fd:  
            return (socket not connected error);  
        case mapped fd:  
            perform mapping to transport fd;  
        case transport fd:  
            if (information is available from the proxy node)  
                return (foreign protocol address associated with this transport fd);  
            else  
                return (address not available);  
        default:  
            return (getpeername (fd, localaddr, addrlen));  
    }  
}
```

FIG. 60

```
getsockopt() → sf_getsockopt (fd, level, optname, optval, optlen) {  
    if (level == SOL_SOCKET) {  
        switch (type of fd) {  
            case service fd:  
                return (warning: setsockopt() not meaningful for service fd);  
            case mapped fd:  
                perform mapping to transport fd;  
            case transport fd:  
                switch (optname) {  
                    case SO_RCVBUF:  
                    case SO_SNDBUF:  
                        if (buffering supported by proxy node) {  
                            get corresponding state variable and place value in optval;  
                            return (success);  
                        }  
                    else  
                        return (unable to get buffer sizes);  
                    case SO_LINGER:  
                    case SO_RCVLOWAT:  
                    case SO_SNDLOWAT:  
                        get corresponding state variable and place value in optval;  
                        return (success);  
                    case SO_TYPE:  
                        return (SOCK_STREAM);  
                    default:  
                        return (warning: option not meaningful in SAN Transport);  
                }  
            }  
        }  
    }  
}
```

FIG. 6P-1

```

default:
    return ( getsockopt(fd, level, optname, optval, optlen) );
}

if (level == IPPROTO_TCP) {
    switch (type of fd) {
        case service fd:
            return (warning: setsockopt() not meaningful for service fd);
        case mapped fd:
            perform mapping to transport fd;
        case transport fd:
            switch (optname) {
                case TCP_MAXSEG:
                    get segment size of SAN transport and place value in optval;
                    return (success);
                case TCP_NODELAY:
                    if (no-delay option is known) {
                        get value and place in optval;
                        return (success);
                    }
                    else
                        return (error);
            }
        default:
            return (warning: option not meaningful in SAN Transport);
    }
}

default:
    return ( getsockopt(fd, level, optname, optval, optlen) );
}
return (not implemented);
}

```

FIG. 6P-2

```

setsockopt() → sf_setsockopt (fd, level, optname, optval, optlen) {
    if (level == SOL_SOCKET) {
        switch (type of fd) {
            case service fd:
                return (warning: setsockopt() not meaningful for service fd);
            case mapped fd:
                perform mapping to transport fd;
            case transport fd:
                switch (optname) {
                    case SO_RCVBUF:
                    case SO_SNDBUF:
                        if (buffering supported by proxy node) {
                            set corresponding state variable to value given by optval;
                            communicate buffer size given by optval to proxy node;
                            if (communication successful)
                                return (success);
                            else
                                return (unable to set buffer size);
                        }
                    else
                        return (unable to set buffer sizes);
                }
            case SO_LINGER:
            case SO_RCVLOWAT:
            case SO_SNDLOWAT:
                set corresponding state variable to value given by optval;
                communicate optname and optval to proxy node;
                if (communication successful)
                    return (success);
                else
                    return (unable to set option);
        }
    }
}

```

FIG. 6Q-1

```

    default:
        return (warning: option not meaningful in SAN Transport);
    }
    default:
        return ( setsockopt(fd, level, optname, optval, optlen) );
    }
}

if (level == IPPROTO_TCP) {
    switch (type of fd) {
        case service fd:
            return (warning: setsockopt() not meaningful for service fd);
        case mapped fd:
            perform mapping to transport fd;
        case transport fd:
            switch (optname) {
                case TCP_MAXSEG:
                    set segment size of SAN transport to value given by optval;
                    return (success);
                case TCP_NODELAY:
                    set no-delay variable to value given by optval;
                    communicate optname and optval to the proxy node;
                    if (communication successful)
                        return (success);
                    else
                        return (unable to set no-delay option);
            }
        default:
            return (warning: option not meaningful in SAN Transport);
    }
}

default:
    return ( setsockopt(fd, level, optname, optval, optlen) );
}
}
return (not implemented);
}

```

FIG. 6Q-2

```
close() → sf_close (fd) {  
    switch (type of fd) {  
        case service fd:  
            send LEAVE_SERVICE msg on service fd;  
            clean up transport resources associated with this service;  
            return (close(fd));  
        case mapped fd:  
            perform mapping to transport fd;  
            send CLOSE_CONNECTION msg on transport fd;  
            reset fd mapping;  
            return (close (fd));  
        case transport fd:  
            send CLOSE_CONNECTION msg on transport fd;  
        default:  
            return (close(fd));  
    }  
}
```

FIG. 6R

```
shutdown() → sf_shutdown (fd, howto) {  
    if (howto == SHUT_RD) {  
        if (fd already closed for writes)  
            set full_shutdown_flag to TRUE;  
        else  
            note that fd is closed for further reads;  
    }  
  
    if (howto == SHUT_WR) {  
        if (fd already closed for reads)  
            set full_shutdown_flag to TRUE;  
        else  
            note that fd is closed for further writes;  
    }  
  
    if (howto == SHUT_RDWR) {  
        set full_shutdown_flag to TRUE;  
    }  
  
    if (full_shutdown_flag == TRUE) {  
        switch (type of fd) {  
            case service fd:  
                send LEAVE_SERVICE msg on service fd;  
                clean up transport resources associated with this service;  
                break;  
  
            case mapped fd:  
                perform mapping to transport fd;  
                send CLOSE_CONNECTION msg on transport fd;  
                reset fd mapping;  
                break;  
  
            case transport fd:  
                send CLOSE_CONNECTION msg on transport fd;  
                break;  
  
            default:  
                return ( shutdown (fd, howto) );  
        }  
    }  
  
    return ( shutdown (fd, howto));  
}
```

FIG. 6S